Python: Beyond The Basics

Aaron Maxwell

aaron@powerfulpython.com

Contents

1 Iterators and Generators

Consider the humble `for` loop:

```
for item in items:
    do_something_with(item)
```

Surprising miracles hide here. You likely know that the act of efficiently going through a collection, one element at a time, is called *iteration*. What's less well understood is just how deep, sophisticated, and just well-thought-out Python's iteration system really is. As you more fully understand how it works, you naturally get the ability to write very scalable Python applications… able to handle larger data sets in performant, memory-efficient ways.

As valuable as this is on its own, understanding iteration also lets you to master another extremely powerful tool: *generator functions*. Generator functions are not only a convenient way to create useful iterators. They enable exquisite patterns of encapsulation and code organization, in a way that - by their fundamental nature - intrinsically encourage some excellent coding habits.

This may be the most important chapter in the book. Understanding it threatens to make you a permanently better programmer, in every language. Let's dive in.

1.1 Iteration in Python

Python has a built-in function called `iter()`. When you pass it a collection, you get back an *iterator object*:

```
>>> numbers = [0, 1, 2, 3]
>>> iter(numbers)
<list_iterator object at 0x101b20c18>
```

Just as in other languages, a Python iterator produces the values in a sequence, one at a time. You probably know an iterator is like a moving pointer over the collection:

```
>>> it = iter(numbers)
>>> for n in it: print(n)
0
1
2
3
```

In Python, you can only scan through an individual iterator once. Once it's produced all elements, that iterator is exhausted, and won't produce anything if you use it again:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> names_it = iter(names)
>>> for name in names_it:
...     print(name)
Tom
Shelly
Garth
>>> for name in names_it:
...     print(name)
>>> print("No output the second time!")
No output the second time!
```

If you *do* need to go through the source collection twice, you can simply create two iterators:

```
>>> names_it1 = iter(names)
>>> names_it2 = iter(names)
>>> for name in names_it1: print(name)
Tom
Shelly
Garth
>>> for name in names_it2: print(name)
Tom
Shelly
Garth
```

When you write `for name in names`, what Python effectively does under the hood is call `iter()` on that collection. Whatever object it gets back is used as the iterator for that `for` loop:

```
for name in names:
    print(name)
# just like:
names_iter = iter(names)
for name in names_iter:
    print(name)
```

Each iterator has its own identity, separate from the source collection - which you can verify with `id()`[1]:

[1] The built-in `id` function takes an object, and returns an integer. That number will always be *different* for

```
>>> names_it1 = iter(names)
>>> names_it2 = iter(names)
>>> id(names)
4323311816
>>> id(names_it1)
4323413464
>>> id(names_it2)
4323413408
```

How does `iter()` actually get the iterator? It can do this in several ways, but the most important relies on a magic method called `__iter__`. This is a method any class (including yours) may define; when called with no arguments, it must return a fresh iterator object. Lists have it, for example:

```
>>> numbers.__iter__
<method-wrapper '__iter__' of list object at 0x10130e4c8>
>>> numbers.__iter__()
<list_iterator object at 0x1013180f0>
```

Python makes a distinction between objects which are *iterators*, and objects which are *iterable*. We say an object is *iterable* if and only if you can pass it to `iter()`, and get a ready-to-use iterator. Often this is because that object has an `__iter__` method, which `iter()` will call. Python lists and tuples are iterable. So are strings, which is why you can write `for char in my_str:` to iterate over `my_str`'s characters. Any container you might use in a `for` loop is iterable.

Of course, a `for` loop is only one way to process the elements of a sequence. Sometimes your code needs to step through in a more fine-grained way. For this, Python provides the built-in function `next`. In normal usage, you call it with a single argument, which is an iterator. Each time you call it, `next(my_iterator)` fetches and returns the next element:

two different objects, and always the *same* for the same object. So you can use `id()` to determine whether two variables store the same object or not. In Python's main implementation, this number currently corresponds to a memory address, but that's not required by the language definition.

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> # Create a fresh iterator...
... names_it = iter(names)
>>> next(names_it)
'Tom'
>>> next(names_it)
'Shelly'
>>> next(names_it)
'Garth'
```

What happens if you call `next(names_it)` again? `next()` will raise a special built-in exception, called `StopIteration`:

```
>>> next(names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

This is part of Python's *iterator protocol*. Raising this specific exception is, by design, how an iterator is supposed to signal that the sequence is done. You rarely have to raise or catch this exception yourself, though we'll see some patterns later where it's useful to do so. A good mental model for how a `for` loop works is to imagine it calling `next()` each time through the loop, exiting when `StopIteration` gets raised.

When using `next()` yourself, you can provide a second argument, for the default value. If you do, `next()` will return that instead of raising `StopIteration` at the end:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> new_names_it = iter(names)
>>> next(new_names_it, "Rick")
'Tom'
>>> next(new_names_it, "Rick")
'Shelly'
>>> next(new_names_it, "Rick")
'Garth'
>>> next(new_names_it, "Rick")
'Rick'
>>> next(new_names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(new_names_it, "Jane")
'Jane'
```

1.2 Beyond Simple Containers

Now, let's consider a different situation. What if you aren't working with a simple sequence of numbers or strings, but something more complex? What if you are calculating or reading or otherwise obtaining the sequence elements as you go along? Let's start with a simple example (so it's easy to reason about). Suppose you need to write a function creating a list of square numbers, which will be processed by other coded:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This works. But there is potential problem lurking here. Can you spot it?

Here's one: what if `MAX` is not 5, but 10,000,000? Or 10,000,000,000? Or more? **The second `for` loop cannot event *start*** until the *entire* list of squares has been fully calculated.

Even worse: What if you aren't doing arithmetic to get each element - which is fast and cheap - but making a truly expensive calculation? Or making an API call over the network? Or reading from a database? Your program is sluggish, even unresponsive, and users will think you're not a good programmer.

The solution is to create an iterator to start with, lazily computing each value only when it's needed. Then each cycle through the loop happens just in time.

For the record, here is how you create an equivalent iterator class, which fully complies with the entire Python iterator protocol:

```
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Holy crap, that's horrible. There's got to be a better way.

Good news: there's a better way. It's called a **generator function**, and you're going to love it!

1.3 Generator Functions

Python provides a tool called the **generator function**, which - among other talents - is a *very* useful shortcut for creating iterators. A generator function looks a lot like a regular function, but instead of saying `return`, it uses a new and different keyword: `yield`. Here's a simple example:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

We can then use it in a for loop like this:

```
>>> for num in gen_nums():
...     print(num)
0
1
2
3
```

Let's go through and understand this. When you call `gen_nums()` like a function, it immediately returns a **generator object**:

```
>>> sequence = gen_nums()
>>> type(sequence)
<class 'generator'>
```

The *generator function* is `gen_nums` - what we define and then call. A function is a generator function if and only if it uses "yield" instead of "return". The *generator object* is what that generator function returns when called (`sequence`, in this case). This generator object is an iterator, which means you can iterate through it using `next()` or a `for` loop:

```
>>> sequence = gen_nums()
>>> next(sequence)
0
>>> next(sequence)
1
>>> next(sequence)
2
>>> next(sequence)
3
>>> next(sequence)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
>>> # Or in a for loop:
... for num in gen_nums(): print(num)
...
0
1
2
3
```

The flow of code works like this: when `next()` is called the first time, or the `for` loop first starts, the body of `gen_nums` starts executing at the beginning, and returns the value to the right of the `yield` to the iterator.

So far, this is much like a regular function. But the next time `next()` is called - or, equivalently, the next time through the `for` loop - the function doesn't start at the beginning again. It starts on the line *after the yield statement*. Look at the source of `gen_nums()` again:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

`gen_nums` is more general than a function or a subroutine. It's actually a *coroutine*. You see, a regular function can have several exit points (otherwise known as `return` statements). But it has only one entry point: each time you call a function, it always starts at the first line of the function body.

A coroutine is like a subroutine, except that it has several possible *entry* points. It starts with first line, like a normal function. But when it "returns", the coroutine isn't exiting, so much as *pausing*. Subsequent calls with `next()` - or equivalently, the next time through the `for` loop - start at that `yield` statement again, right where it left off; the re-entry point is the line after the `yield` statement.

For Python generator functions, each time a new value is requested, the flow of control picks up on the line after the `yield` statement. In this case, the next line increments the variable `n`, then continues with the while loop.

Notice we do not raise `StopIteration` anywhere in the body of `gen_nums()`. When the function body finally exits - after it exits the while loop, in this case - the generator object automatically raises `StopIteration`.

Again: each `yield` statement simultaneously defines an exit point, *and* a re-entry point. In fact, you can have multiple `yield` statements in a generator:

```
def gen_extra_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
    yield 42 # Second yield
```

Here's the output when you use it:

```
>>> for num in gen_extra_nums():
...     print(num)
0
1
2
3
42
```

The second `yield` is reached after the `while` loop exits. When the function reaches the implicit return at the end, the iteration stops. Reason through the code above, and convince yourself it makes sense.

Let's revisit the earlier example, of cycling through a sequence of squares. This is how we first did it:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

As an exercise, pause here, open up a new Python file, and see if you can write a gen_squares generator function that accomplishes the same thing.

Done? Great. Here's what it looks like:

```
>>> def gen_squares(max_num):
...     for num in range(max_num):
...         yield num ** 2
...
>>> MAX = 5
>>> for square in gen_squares(MAX):
...     print(square)
0
1
4
9
16
```

Now, this gen_squares has a problem in Python 2, but not in Python 3. Can you spot it?

Here it is: range returns an iterator in Python 3, but in Python 2 it returns a list. If MAX is huge, that creates a huge list inside, which obviously destroys scalability. So if you are using Python 2, your gen_squares needs to use xrange instead, which acts just like Python 3's range.

The larger point here affects all versions of Python. Generator functions *potentially* have a small memory footprint, but only if you code intelligently. When writing generator functions, be watchful for hidden bottlenecks.

Now, strictly speaking, we don't *need* generator functions. We just *want* them, because they make certain useful patterns far easier. Now that we're in a position to understand it, let's look at the SquaresIterator class again:

```python
# Same code we saw earlier.
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Each value is obtained by invoking its `__next__` method, until it raises `StopIteration`. This produces the same output; But look at the source for the `SquaresIterator` class, and compare it to the source for the generator above. Which is easier to read? Which is easier to maintain? And when requirements change, which is easier to modify without introducing errors? Most people find the generator solution easier and more natural.

Authors often use the word "generator" by itself, to mean either the generator function, *or* the generator object returned when you call it. Typically the writer thinks it's obvious by the context which they are referring to; sometimes it is, sometimes not. Sometimes the writer is not even clear on the distinction to begin with. But it's important: just as there is a big difference between a function, and the value it returns when you call it, so is there a big difference between the generator function, and the generator object it returns.

In your own thought and speech, I encourage you to only use the phrases "generator function" and "generator object", so you are always clear inside yourself, and in your communication. (Which also helps your teammates be more clear.) The only exception: when you truly mean "generator functions and objects", lumping them together, then it's okay to just say "generators". I'll lead by example in this book.

1.4 Python is Filled With Iterators

Let's take a look at Python 3 dictionaries:[2]

```
>>> calories = {
...     "apple": 95,
...     "slice of bacon": 43,
...     "cheddar cheese": 113,
...     "ice cream": 15, # You wish!
... }
>>> items = calories.items()
>>> type(items)
<class 'dict_items'>
```

So what is this `dict_items` object returned by `calories.items()`? It turns out to be what Python calls a *view*. There is not any kind of base view type; rather, an object quacks like a dictionary view if it supports three things:

- `len(view)` returns the number of items,
- `iter(view)` returns an iterator over the key-value pairs, and
- `(key, value) in view` returns True if that `key-value` pair is in the dictionary, else False.

In other words, a dictionary view is iterable, with a couple of bonus features. It also dynamically updates if its source dictionary changes:

```
>>> items = calories.items()
>>> len(items)
4
>>> calories['orange'] = 50
>>> len(items)
5
>>> ('orange', 50) in items
True
>>> ('orange', 20) in items
False
```

[2] If you're more interested in Python 2, follow along. Every concept in this section fully applies to Python 2.7, with syntax differences we'll discuss at the end.

Dictionaries also have `.keys()` and `.values()`. Like `.items()`, they each return a view. But instead of key-value pairs, they only contain keys or values, respectively:

```
>>> foods = calories.keys()
>>> counts = calories.values()
>>> 'yogurt' in foods
False
>>> 100 in counts
False
>>> calories['yogurt'] = 100
>>> 'yogurt' in foods
True
>>> 100 in counts
True
```

In Python 2[3], `items()` returns a list of key-value tuples, rather than a view; Python 2 also has an `iteritems()` method that returns an iterator (rather than an iterable view object). Python 3's version of the `items` method essentially obsoletes both of these. If you truly need a list of key-value pairs in Python 3, you can always just write `list(calories.items())`.

Iteration has snuck into many places in Python. The built-in `range` function returns an iterable:

```
>>> seq = range(3)
>>> type(seq)
<class 'range'>
>>> for n in seq: print(n)
0
1
2
```

The built-in `map`, `filter`, and `zip` functions all return iterators:

[3] See next section for details.

```
>>> numbers = [1, 2, 3]
>>> big_numbers = [100, 200, 300]
>>>
>>> def double(n): return 2 * n
...
>>> def is_even(n): return n % 2 == 0
...
>>> mapped = map(double, numbers)
>>> mapped
<map object at 0x1013ac518>
>>> for num in mapped: print(num)
...
2
4
6
>>> filtered = filter(is_even, numbers)
>>> filtered
<filter object at 0x1013ac668>
>>> for num in filtered: print(num)
...
2
>>> zipped = zip(numbers, big_numbers)
>>> zipped
<zip object at 0x1013a9608>
>>> for pair in zipped: print(pair)
...
(1, 100)
(2, 200)
(3, 300)
```

Notice that `mapped` is something called a "map object", rather than a list of the results of the calculation; and similar for `filtered` and `zipped`. This gives you all the benefits of iteration, built in to the language.

1.4.1 Python 2's Differences

For Python 2, I'll start with some recommendations (which I explain in detail below):

- With dictionaries, always use `viewitems()` rather than `items()` or `iteritems()`. The only exception: if you truly need a list of tuples, use `items()`.

- Likewise for `viewkeys()` and `viewvalues()`, rather than `keys()`, `iterkeys()`, `values()`, and `itervalues()`.
- Use `xrange()` instead of `range()`, unless you have a special need for an actual list.
- Be aware that `map`, `filter`, and `zip` create lists; if the data may need to scale to arbitrarily larger sizes, use `imap`, `ifilter` or `izip` from the `itertools` module instead.

These differently named methods and functions essentially have the same behavior of Python 3's versions above. Python 2's `xrange` is just like Python 3's `range`; Python 2's `itertools.imap` is just like Python 3's `map`; and so on.

Let's examine Python 2's dictionary methods. In Python 2, `calories.items()` returns a list of `(key, value)` tuples. So if the dictionary has 10,000 keys, you'd get a list of 10,000 tuples. Similarly, `calories.keys()` returns a list of keys; `calories.values()` returns a list of values. The problems with this will be obvious to you by now: the loop blocks until you create and populate a list, which is immediately thrown away once the loop exits.

Python 2 addressed this by introducing two other methods: `iteritems()`, returning an iterator over the key-value tuples; and (later) `viewitems()`, which returned a view - an iterable type. Similarly, `keys()` gave a list of keys, and they added `iterkeys()` and then `viewkeys()`; and again for `values()`, `itervalues()`, and `viewvalues()`.

In Python 3, what used to be called `viewitems()` was renamed `items()`, and the old `items()` and `iteritems()` went away. Similarly, `keys()` and `values()` were changed to return views instead of lists.

For your own Python 2 code, I recommend you start using `viewitems()`, except when you have an explicit reason to do otherwise. Using `iteritems()` is certainly better than using `items()`, and for Python 2 code, generally works just as well as `viewitems()`. However, if you ever decide to upgrade that codebase with `2to3`, the resulting code will be closer to your original program.[4] Python 2's `viewitems` basically obsoleted `iteritems`, which is why the latter has no equivalent in Python 3.

The situation with `range` is simpler. Python 2's original `range()` function returned a list; later, `xrange()` was added, returning an iterable, and practically speaking obsoleting Python 2's `range()`. But many people continue to use `range()`, for a range (ha!) of reasons. Python

[4] `2to3` will replace both `iteritems()` and `viewitems()` with `items()`; but the precise semantics of the converted program will more closely match your Python 2 code if you use `viewitems()` to begin with.

3's version of `range()` is essentially the same as Python 2's `xrange()`, and Python 3 has no function named `xrange`. (Of course, `list(range(. . . ))` will give you an actual list, if you need it.)

`map`, `filter`, and `zip` are widely used in data science, among other fields. If you want your Python 2 code using these functions to be fully forward-compatible, you have to go to a little more trouble: their iterator-equivalents are all in the `itertools` module. So, instead of this:

```
mapped = map(double, numbers)
```

you will need to write this:

```
from itertools import imap
mapped = imap(double, numbers)
```

The `2to3` program will convert Python 2's `imap(f, items)` to `map(f, items)`, but will convert Python 2's `map(f, items)` to `list(map(f, items))`. The `itertools` module similarly has `ifilter` and `izip`, for which the same patterns apply.

It's important to realize that everything described for Python 3 also applies to Python 2.7, *if* you use the different names of the relevant methods and functions. And that is what I recommend you do, so you get the scalability benefits of iterators, and have an easier transition to Python 3.

1.5 Python's Iterator Protocol

This optional section explains Python's **iterator protocol** in formal detail, giving you a more precise and fundamental understanding of how generators, iterators, and iterables all work. For the day-to-day coding of most programmers, it's not nearly as important as everything else in this chapter. That said, you need this information to implement your own, custom iterable collection types. Personally, I also find knowing the protocol helps me reason more easily about iteration-related bugs and edge cases. If this all sounds valuable to you, keep reading; otherwise, feel free to skip to the next chapter.

As mentioned, Python makes a distinction between *iterators*, versus objects that are *iterable*. The difference is subtle to begin with, and frankly it doesn't help that the two words sound nearly identical. Keep clear in your mind that "iterator" and "iterable" are distinct but related things, and the following will be easier to understand.

Informally, an iterator is something you can pass to `next()`, or use exactly once in a `for` loop. More formally, an object in Python 3 is an iterator if it meets the following criteria:

- It defines a method named `__next__`, which may be called with no arguments.
- Each time `__next__()` is called, it produces the next item in the sequence.
- Until all items have been produced. Then, subsequent calls to `__next__()` raise `StopIteration`.
- The iterator must also define a boilerplate method named `__iter__`, called with no arguments, and returning the same iterator. Its body is literally `return self`.

Any object with these methods can call itself a Python iterator. You are not intended to call the `__next__()` method directly yourself. Instead, you will use the built-in `next()` function. In fact, here is a simplified[5] way you might implement `next()` yourself:

```
def next(it, default=None):
    try:
        return it.__next__()
    except StopIteration:
        if default is None:
            raise
        return default
```

All the above has one difference in Python 2: the iterator's method is named `.next()` rather than `.__next__()`. Abstracting over this difference is one reason to use the built-in, top-level `next()` function. Of course, using `next()` lets you specify a default value, whereas `.__next__()` and `.next()` do not.

Now, all the above is for the "iterator". Let's explain the other word, "iterable". Informally, an object is *iterable* if it knows how to create an iterator over its contents, which you can access with the built-in `iter()` function. More formally, a Python container object is *iterable* if it meets one of these two criteria:

- It defines a method called `.__iter__()`, which creates and returns an iterator over the elements in the container; **or**
- it follows the *sequence protocol*. This essentially means it defines `__getitem__`, i.e. the magic method for square brackets, and lets you reference `foo[0]`, `foo[1]`, etc., raising an `IndexError` once you go past the last element.

[5] This version will not let you specify `None` as a default value; the real built-in `next()` will. Otherwise, this simplified version is essentially accurate.

Generally, if you are implementing your own container type, you will want to implement `__iter__`. You can return a generator object from this method.

Notice now the relationship between the `__iter__` methods of iterators and iterables. Both are called with no argument, and both return an iterator object. The only difference is that the iterator returns its `self`, while an iterable will create and return a *new* iterator. (And if you call it twice, it will give you two different iterators.)

This similarity is intentional, to simplify `for` loops and other control code that can accept either iterators or iterables. Here's the mental model you can safely follow: when Python's runtime encounters a `for` loop, it will start by invoking `iter(sequence)`. This *always* returns an iterator: either `sequence` itself, or (if `sequence` is only iterable) the iterator created by `sequence.__iter__()`.

Iterables are everywhere in Python. Almost all built-in collection types are iterable:

```
>>> # A list is not an iterator...
... items = [7, 2, 3]
>>> hasattr(items, '__next__')
False
>>> # But a list IS iterable.
... hasattr(items, '__iter__')
True
>>> # Which means we can create an iterator object from it.
... items_iter = iter(items)
>>> # This works in a for loop.
... for item in items_iter: print(item)
7
2
3
>>> # But for-loops wrap their argument in iter() automatically,
... # so we don't normally need to do that.
... for item in items: print(item)
7
2
3
```

In your own custom collection classes, sometimes the easiest way to implement `__iter__()` actually involves using `iter()`. For instance, this will not work:

```
class BrokenInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        return self.items
```

If you try it, you get a `TypeError`:

```
>>> items = BrokenInLoops()
>>> for item in items:
...     print(item)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: iter() returned non-iterator of type 'list'
```

Quick quiz: what one-line change can you make to `BrokenInLoops` that will make it truly iterable?

Answer: Use `iter()` inside `__iter__()`:

```
class WorksInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        return iter(self.items)
```

This makes `WorksInLoops` itself be iterable, because `__iter__` now follows the iterator protocol correctly. Each time it's called it generates a fresh iterator.

```
>>> items = WorksInLoops()
>>> for item in items:
...     print(item)
7
3
9
```

Now, even though `WorksInLoops` itself is iterable, it is not an *iterator*. To do that, we must change `__iter__`, and provide a `__next__` method. In that case, we do not need to use `iter()` at all:

```
class WorksAsIterator:
    def __init__(self):
        self.items = [3, 9, 2]
    def __iter__(self):
        return self
    def __next__(self):
        if self.items:
            return self.items.pop()
        raise StopIteration
```

Implemented this way, we can use it simply in a for loop:

```
>>> items = WorksAsIterator()
>>> for item in items:
...     print(item)
7
3
9
```

Note that in the code for `WorksAsIterator` above, `__iter__` just returns `self`. That's a common pattern when implementing classes that follow the iterator protocol. (Internally Python requires iterators to implement `__iter__`, even if it just returns `self` - so we always have to do it.)

In practice, you almost never invoke `__iter__` directly, instead letting Python do so implicitly. You will also never directly invoke `__next__`, but for a different reason - because we have a built-in function called `next()` that we use instead:

```
>>> items = WorksAsIterator()
>>> next(items)
2
>>> next(items)
9
>>> next(items)
3
>>> next(items)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __next__
StopIteration
```

Here, `next()` is getting the next value by calling `items.__next__()`, allowing `StopIteration` to raise at the end. Often you don't need to fetch the next item directly, but sometimes you do, and using `next(iterator)` is preferable to `iterator.__next__()` for several reasons. For one thing, it's easier to type and read. Also, `next()` lets you provide a default value as the second argument:

```
>>> next(items)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __next__
StopIteration
>>> next(items, 42)
42
>>> next(items)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __next__
StopIteration
>>> next(items, 99)
99
```

This gives you an alternative way to handle an iterator being exhausted.

2 Creating Collections with Comprehensions

A *list comprehension* is a high level, declarative way to create a list in Python. They look like this:

```
>>> squares = [ n*n for n in range(6) ]
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

This is exactly equivalent to the following:

```
>>> squares = []
>>> for n in range(6):
...     squares.append(n*n)
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice that in the first example, what you type is declaring *what* kind of list you want, while the second is specifying *how* to create it. That's why we say it is high-level and declarative: it's as if you are stating what kind of list you want created, and then let Python figure out how to build it.

Python lets you write other kinds of comprehensions other than lists. Here's a simple dictionary comprehension, for example:

```
>>> blocks = { n: "x" * n for n in range(5) }
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

This is exactly equivalent to the following:

```
>>> blocks = dict()
>>> for n in range(5):
...     blocks[n] = "x" * n
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

The benefits of comprehensions primarily have to do with readability and maintainability. Most people find them *very* readable; even developers who have never encountered them before can usually correctly guess what it means. There is also a cognitive benefit: once you've practiced with them a bit, you will find you can write them with minimal mental effort - keeping more of your attention free for other tasks.

Beyond lists and dictionaries, there are several other forms of comprehension you will learn about it in this chapter. As you become comfortable with them, you will find them to be versatile and very Pythonic - meaning, you'll find they fit well into many other Python idioms and constructs, lending new expressiveness and elegance to your code.

2.1 List Comprehensions

A list comprehension is the most widely used and useful kind of comprehension, and is essentially a way to create and populate a list. Its structure looks like:

```
[ EXPRESSION for VARIABLE in SEQUENCE ]
```

EXPRESSION is any Python expression, though in useful comprehensions, the expression typically has some variable in it. That variable is stated in the *VARIABLE* field. *SEQUENCE* defines the source values the variable enumerates through, creating the final sequence of calculated values.

Here's the simple example we glimpsed earlier:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice the result is just a regular list. In `squares`, the expression is `n*n`; the variable is `n`; and the source sequence is `range(6)`. Note the sequence is a `range` object; in general, it could in fact be any iterable… an actual list or tuple, a generator object, or something else.

The expression part can be anything that reduces to a value:

- Arithmetic expressions like `n+3`
- A function call like `f(m)`, using `m` as the variable
- A slice operation (like `s[::-1]`, to reverse a string)
- Method calls (`foo.bar()`, iterating over a sequence of objects)

• and more.

Some complete examples:

```
>>> # First define some source sequences...
... pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> # And a helper function...
... def repeat(s):
...     return s + s
...
>>> # Now, some list comprehensions:
... [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]
>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]
>>> [ 10 - x for x in numbers ]
[1, 11, 14, -10, -1, 13]
>>> [ pet.lower() for pet in pets ]
['dog', 'parakeet', 'cat', 'llama']
>>> [ "The " + pet for pet in sorted(pets) ]
['The cat', 'The dog', 'The llama', 'The parakeet']
>>> [ repeat(pet) for pet in pets ]
['dogdog', 'parakeetparakeet', 'catcat', 'llamallama']
```

Notice how all these fit the same structure. They all have the keywords "for" and "in"; those are required in Python, for any kind of comprehension you may write. These are interleaved among three fields: the expression; the variable (i.e., the identifier from which the expression is composed); and the source sequence.

And of course, the order of elements in the final list is determined by the order of the source sequence. But you can filter out elements by adding an "if" clause:

```
>>> def is_palindrome(s):
...     return s == s[::-1]
...
>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> words = ["bib", "bias", "dad", "eye", "deed", "tooth"]
>>>
>>> [ n*2 for n in numbers if n % 2 == 0 ]
[-8, 40]
>>>
>>> [pet.upper() for pet in pets if len(pet) == 3]
['DOG', 'CAT']
>>>
>>> [n for n in numbers if n > 0]
[9, 20, 11]
>>>
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

The structure is

```
[ EXPR for VAR in SEQUENCE if CONDITION ]
```

where *CONDITION* is an expression that evaluates to `True` or `False`, depending on the variable.[6] Note that it can be either a function applied to the variable (`is_palindrome(word)`), or something more complex (`len(pet) == 3`). Choosing to use a function can improve readability, and also let you apply filter logic whose code won't fit in one line.

A list comprehension must always have the "for" word, even if the beginning expression is just the variable itself. For example, when we say:

```
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

Sometimes people think `word for word in words` is redundant[7], and try to shorten it… but that doesn't work:

[6] Technically, the condition doesn't have to depend on the variable, but it's hard to imagine building a useful list comprehension this way.

[7] It is.

```
>>> [word in words if is_palindrome(word)]
  File "<stdin>", line 1
    [word in words if is_palindrome(word)]
                                         ^
SyntaxError: invalid syntax
>>>
```

You can have several `for VAR in SEQUENCE` clauses. This lets you construct longer lists by doing a kind of cross-product between two or more sources:

```
>>> colors = ['orange', 'purple', 'pink']
>>> things = ['truck', 'hat', 'book']
>>> colorful_things = [ color + ' ' + thing
...                     for color in colors
...                     for thing in things ]
>>> for item in colorful_things:
...     print(item)
...
orange truck
orange hat
orange book
purple truck
purple hat
purple book
pink truck
pink hat
pink book
```

Notice you can insert newlines inside a list comprehension - and you should do so freely in your programs, to make the expression more readable. Python's normal whitespace/indentation rules are essentially suspended inside the square brackets, making it easy to split up long lines. A good general structure is

```
def get_list(some_args):
    # ....
    return [ EXPR
             for VAR in SEQUENCE
       if CONDITION ]
```

Some people prefer to put the brackets on their own lines, and/or put the expression and first for-clause together:

```
def get_list(some_args):
    # ....
    return [
        EXPR for VAR in SEQUENCE
        for OTHER_VAR in OTHER_SEQUENCE
  ]
```

Giving brackets their own lines has the advantage that you don't need to indent the initial expression as much, nor mess up your nice alignment if you change anything to the left of the opening bracket. On the other hand, it takes up vertical space. Experiment to find what you find most readable and easy to work with.footnote:

You can use multiple `if` clauses, just like with the `for` clause:

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> odd_positives = [
...     num for num in numbers
...     if num > 0
...     if num % 2 == 1
... ]
>>> print(odd_positives)
[9, 11]
```

And of course, you can use multiple `for` and `if` clauses together. The only rule is that the first `for` clause must come before the first `if` clause. Generally, a nicely readable list comprehension will group all the `for` clauses together, followed by all the `if` clauses.

2.2 Generator Expressions

List comprehensions create lists:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
```

When you need a list, that's great, but sometimes you don't *need* a list. In a for loop, for example, anything iterable will suffice. Often it doesn't matter much, but imagine the following:

```
NUM_SQUARES = 10*1000*1000
many_squares = [ n*n for n in range(NUM_SQUARES) ]
for number in many_squares:
    do_something_with(number)
```

Before the very first iteration through the for loop can even start, the entire `many_squares` list must be created. All memory for it must be allocated, and every element must be calculated. This has to happen before `do_something_with` is called even *once*. Here's a self-profiling program little program, that tells us just how long it takes:

```
import datetime
now = datetime.datetime.now
start = now()
# Helper printing how many seconds have passed
# since the program started.
def show_time():
    diff = (now() - start).total_seconds()
    print("Time since start: {:.1f} sec".format(diff))
# Pretend this function does something useful.
def do_something_with(item):
    pass # Stretching the definition of "something".
# First time check...
show_time()
NUM_SQUARES = 10*1000*1000
many_squares = [ n*n for n in range(NUM_SQUARES) ]
# Second time check...
show_time()
for number in many_squares:
    do_something_with(number)
# ... and the time at the end
show_time()
```

Running on my machine produces the output:

```
Time since start: 0.0 sec
Time since start: 0.9 sec
Time since start: 2.2 sec
```

Over 40% of the entire program run time happens before the for loop even starts. And that's for a cheap arithmetic computation. What if the elements come from a high-latency network query?

We can immediately make our program more responsive by using a *generator expression*. Syntactically, it looks very much like a list comprehension. The only difference is that we use parenthesis instead of square brackets:

```
>>> generated_squares = ( n*n for n in range(NUM_SQUARES) )
>>> type(generated_squares)
<class 'generator'>
```

What's this `generator` type? It produces an iterator over the computed values, lazily - one at a time. That's the difference between a generator expression and a list comprehension: the first one computes values lazily, while the second greedily computes everything at the start. We can fetch the next value with the `next` built-in:

```
>>> next(generated_squares)
0
>>> next(generated_squares)
1
>>> next(generated_squares)
4
>>> next(generated_squares)
9
```

More often, though, you will use a generator expression in a for loop. Running the above self-profiling program again, replacing `many_squares` with `generated_squares`, gives different timing:

```
Time since start: 0.0 sec
Time since start: 0.0 sec
Time since start: 2.2 sec
```

The for loop starts almost immediately when we run the program. The act of *defining* a generator is very cheap, in every sense, because the work of producing the sequence items is put off until later.

Generator expressions are sometimes called "generator comprehensions". I actually think that's a better name; after all, it's just another form of comprehension. However, the Python online docs use the term "generator expression", so that's what I will call it here.

Every list comprehension can be converted to a generator expression. They can use one or more `for` clauses, `if` clauses, etc. As the programmer, you only need to type parentheses instead of square brackets. In fact, sometimes it's even more concise than that. When generator expressions are passed in certain contexts, you can omit even the surrounding parenthesis. For example, suppose you are sorting a list of customer email addresses, looking at only those customers whose status is "active". You may have a full list to start with:

```
>>> # User is a class that has an email field.
... # all_users is a list of User objects.
...
>>> first_user = all_users[0]
>>> first_user.email
'fred@a.com'
>>> first_user.is_active
True
```

If I wanted to get a sorted list of the active user's emails, I could pass a list comprehension to `sorted`:

```
>>> # Realistic data would be a much longer list,
... # but this demonstrates the idea.
...
>>> sorted([user.email for user in all_users
...         if user.is_active])
['fred@a.com', 'sandy@f.net', 'tim@d.com']
```

This code creates and populate a list, passes it to `sorted`, then immediately throws that temporary list away. This waste can be avoided with a generator generator expression:

```
>>> # The same, with a generator expression.
... sorted( (user.email for user in all_users
...          if user.is_active) )
['fred@a.com', 'sandy@f.net', 'tim@d.com']
```

That's completely valid Python. But when we pass a generator comprehension in as an argument to a function, we can omit the extra pair of parenthesis completely:

```
>>> # An entirely equivalent generator expression.
>>> sorted(user.email for user in all_users
...        if user.is_active)
['fred@a.com', 'sandy@f.net', 'tim@d.com']
```

Notice how readable and natural this is. At least, it will be after you practice it a little. One thing to watch out for: you can only inline a generator expression this way when passed to a function or method of one argument. Otherwise, you get a syntax error:

```
>>> # Reverse that list. Whoops...
... sorted(user.email for user in all_users
...         if user.is_active, reverse=True)
  File "<stdin>", line 2
SyntaxError: Generator expression must be parenthesized if not ↩
    sole argument
```

Since that generator expression is *not* the sole argument, Python can't unambiguously interpret what you mean, so you must use the parentheses.

```
>>> # Okay, THIS will get the reversed list.
... sorted((user.email for user in all_users
...         if user.is_active), reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

And of course, sometimes it's more readable to assign the generator expression to a variable:

```
>>> active_emails = (
...        user.email for user in all_users
...        if user.is_active
... )
>>>
... sorted(active_emails, reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

Generator expressions without parentheses suggest a unified way of thinking about comprehensions, which link generator expressions and list comprehensions together. Here's a generator expression for a sequence of squares:

```
( n**2 for n in range(10) )
```

And here it is again, passed to the built-in `list()` function:

```
list( n**2 for n in range(10) )
```

And here it is as a list comprehension:

```
[ n**2 for n in range(10) ]
```

When you understand generator expressions, it's easy to see list comprehensions as a derivative data structure. And the same applies for dictionary and set comprehensions, covered in a later section. Once you get this insight, you will start seeing more and more opportunities to use all of them in your own code, improving its readability, maintainability, and performance in the process.

2.2.1 Generator Expression or List Comprehension?

If generator expressions are so great, why would you use list comprehensions? Generally speaking, when deciding which to use, your code will be more scalable and responsive if you use a generator expression, unless you have a reason you need a list comprehension instead. There are several considerations.

First, if the sequence is unlikely to be very big - and by big, I mean many tens of thousands of elements long - there's little benefit in using a generator expression, because they can only be iterated through, and only once at that.[8] They're also immutable. If you need random access, or to go through the sequence twice, or may need to append or remove elements, generator expressions won't work.

This is especially important when writing methods or functions whose return value is a sequence. Do you return a generator, or a list comprehension? There are two schools of thought, and which you choose depends in part on how well your teammates or library users know Python. In theory, there's no reason to ever return a list instead of a generator; a list can be trivially created by passing the generator to `list()`. In practice, the interface may be such that the caller will really want an actual list. Also, if you are constructing the return value as a list within the function, it's silly to return a generator expression over it - just return the actual list.

Also, if your intention is to create a library used mainly by Python novices, that may be an argument for returning a list. After all, almost all programmers are familiar with a list/vector/array data structure, but may get confused when confronted with a generator.

2.3 Dictionaries, Sets, and Tuples

At the beginning of the chapter, you saw an example of a dictionary comprehension. Just like a list comprehension creates a list, a dictionary comprehension creates a dictionary. Here's

[8] If you try to use the same generator expression in two for loops, it will work normally in the first one, but act like an empty list in the second. This is because it's an iterator, but not iterable.

another example. Suppose you have this `Student` class:

```
class Student:
    def __init__(self, name, gpa, major):
        self.name = name
        self.gpa = gpa
        self.major = major
```

Given a list `students` of student objects, we can write a dictionary comprehension mapping student names to their GPAs:

```
>>> { student.name: student.gpa for student in students }
{'Jim Smith': 3.6, 'Ryan Spencer': 3.1,
 'Penny Gilmore': 3.9, 'Alisha Jones': 2.5,
 'Todd Reynolds': 3.4}
```

Notice the syntactic differences compared to list comprehensions. First, we are using curly braces instead of square brackets - which makes complete sense. The second difference is that initial expression is a key-value pair, separated by a colon. So the structure is

```
{ KEY : VALUE for VARIABLE in SEQUENCE }
```

If you prefer, you can use `dict()` instead of curly braces, passing the key-value pairs as a tuple:[9]

```
>>> # The same, using dict() instead of curly braces.
... dict( (student.name, student.gpa) for student in students )
{'Jim Smith': 3.6, 'Ryan Spencer': 3.1,
 'Penny Gilmore': 3.9, 'Alisha Jones': 2.5,
 'Todd Reynolds': 3.4}
```

Of course, you can also use `if` clauses, and do most of the other things you can do with list comprehensions too:

[9] Look closely, and you'll realize this is actually a generator expression being passed to the `dict()` built-in!

```
>>> def invert_name(name):
...     first, last = name.split(" ", 1)
...     return last + ", " + first
...
>>> # Get "lastname, firstname" of the students on the
... # honor's list.
... { invert_name(student.name): student.gpa
...   for student in students
...   if student.gpa > 3.5 }
{'Smith, Jim': 3.6, 'Gilmore, Penny': 3.9}
```

If you've used Python sets before, by now you can probably guess the syntax for set comprehensions. It's exactly like a list comprehension, but instead of square brackets, uses curly braces:

```
>>> # A list of student majors...
... [ student.major for student in students ]
['Computer Science', 'Economics', 'Computer Science', 'Economics' ↩
    , 'Basket Weaving']
>>> # And the same as a set:
... { student.major for student in students }
{'Economics', 'Computer Science', 'Basket Weaving'}
>>> # You can also use the set() built-in.
... set(student.major for student in students)
{'Economics', 'Computer Science', 'Basket Weaving'}
```

You can also make a tuple comprehension, though in my experience they seem to be rarely used. Since the parenthesis syntax is already taken for generator expressions, you must use the `tuple()` built-in:

```
>>> tuple(student.gpa for student in students
...       if student.major == "Computer Science")
(3.6, 2.5)
```

The comprehensions for dicts, sets and tuples are fairly straightforward derivatives of a list comprehension. They differ in that they do not have a generator analogue. If you need to lazily generate key-value pairs or unique elements, your best bet is to implement a generator function.

2.4 Limitations of Comprehensions

Comprehensions have a few wrinkles people sometimes trip over. Consider the following code:

```
# Read in the lines of a file, stripping leading and trailing
# whitespace, and skipping any empty or whitespace-only lines.
trimmed_lines = []
for line in open('wombat-story.txt'):
    line = line.strip()
    if line != "":
        trimmed_lines.append(line)

print("Got {} lines".format(len(trimmed_lines)))
```

Straightforward enough - we're building a list named `trimmed_lines`. The resulting list has all leading and trailing whitespace removed from its elements, skipping any empty lines (or lines that were just whitespace to begin with). It's not hard to imagine needing to do something like this in a real program.

Now… how would you do this using a list comprehension? Here's a first try:

```
with open('wombat-story.txt') as story:
    trimmed_lines = [
        line.strip()
        for line in story
        if line.strip() == ""
  ]

print("Got {} lines".format(len(trimmed_lines)))
```

This works. Notice, though, that `line.strip()` appears twice. That's wasting CPU cycles compared to the for-loop version, which only calls `line.strip()` once. Stripping whitespace from a string isn't *that* expensive, computationally speaking. But sooner or later, you will want to do something where this matters:

```
>>> values = [
...     expensive_function(n)
...     for n in range(BIG_NUMBER)
...     if expensive_function(n) > 0
...     ]
```

So how can you create this as a list comprehension, while calling `expensive_function` only once? It turns out there is no clean way to do this. There overly clever solutions, such as memoizing (which can easily overuse memory), nesting a generator expression inside (which quickly gets unreadable), or making an intermediate list. If the sequence you need fits the pattern above, I recommend you simply do it the old-fashioned way, using a for loop or a generator function.

If you *really* want to use a comprehension, the best approach is to use an intermediate generator expression. The result is fairly readable and understandable:

```
>>> intermediate_values = (
...     expensive_function(n)
...     for n in range(10000)
... )
>>>
>>> values = [
...     intermediate_value
...     for intermediate_value in intermediate_values
...     if intermediate_value > 0
...     ]
```

Another limitation is that comprehensions must be built on one element at a time. The best way to see this is to imagine a list composed of inlined key-value pairs - flattened, in other words, so even-indexed elements are keys, and each key's value comes right after it. Imagine a function that converts this to a dictionary:

```
>>> # Price per pound of various fruits & vegetables, in dollars.
... prices_flat_list = [
...     "orange", 0.70, "banana", 0.86,
...     "cantaloupe", 0.63, "bok choy", 1.56,
...     "coconuts", 1.06 ]

>>> list2dict(prices_flat_list)
{'banana': 0.86, 'bok choy': 1.56, 'cantaloupe': 0.63, 'orange': ↩
   0.7, 'coconuts': 1.06}
```

Here's one way to implement `list2dict`:

```
# Converts a "flattened" list into an "unflattened" dict.
def list2dict(flattened):
    assert len(flattened) % 2 == 0,
        "Input must be list of key-value pairs"
    unflattened = dict()
    for i in range(0, len(flattened), 2):
        key, value = flattened[i], flattened[i+1]
        unflattened[key] = value
    return unflattened
```

Look at `list2dict`'s for loop. It runs through the even index numbers of elements in `flattened`, rather than the elements of `flattened` directly. This allows it to refer to two different list elements each time through the for loop. But this turns out to be something which just can't be expressed in the semantics of a Python comprehension. Generally, a comprehension operates by looking at each element in some source sequence, one at a time; it can't peek at neighboring elements. (Or at pairs of elements, in the case of two `for` clauses, etc.) Another example would be a function that groups the elements of a sequence by some criteria - for example, the first letter of a string:

```
>>> names = ["Joe", "Jim", "Todd",
...     "Tiffany", "Zelma", "Gerry", "Gina"]
>>> grouped_names = group_by_first_letter(names)
>>> grouped_names['j']
['Joe', 'Jim']
>>> grouped_names['z']
['Zelma']
```

Here's one way to implement the grouping function:

```
from collections import defaultdict
def group_by_first_letter(items):
    grouped = defaultdict(list)
    for item in items:
        key = item[0].lower()
        grouped[key].append(item)
    return grouped
```

Again, the semantics of Python comprehensions aren't built to support this kind of algorithm. In functional programming terms, comprehensions can use map and filter operations, but not reduce or fold. Fortunately, this covers many use cases. I point out these limitations to help you avoid wasting time trying to figure them out; in spite of them, I find comprehensions to be a valuable part of my daily Python toolbox, and believe you will too.

3 Classes and Objects: Beyond The Basics

This chapter assumes you are familiar with Python's OOP basics: creating classes, defining methods, and using inheritance. We build on this.

As with any object-oriented language, it's useful to learn about **design patterns** - reusable solutions to common problems involving classes and objects. A LOT has been written about design patterns. Curiously, though, much of what's out there doesn't completely apply to Python - or, at least, it applies *differently*.

That's because many of these design-pattern books, articles, and blog posts are for languages like Java, C++ and C#. But as a language, Python is quite different. Its dynamic typing, first-class functions, and other additions all mean the "standard" design patterns just work differently.

So let's learn what Pythonic OOP is *really* about.

3.1 Quick Note on Python 2

This chapter uses Python 3 syntax. Python 2.7 is nearly the same, and I'll point out the minor differences as we go along. But there is one *big* difference worth emphasizing here.

In modern Python, all classes need to inherit from a built-in base class called `object`. (It's lowercased, defying the normal convention.) This happens automatically for all classes in Python 3:

```
>>> # Python 3
... class Dog:
...     def speak(self):
...         return "woof"
...
>>> dog = Dog()
>>> isinstance(dog, object)
True
```

In Python 2, you must explicitly inherit your classes from `object`. Fail to do this, and your class builds on "old-style classes":

```
>>> # Python 2
... class DogFromObject(object):
...     def speak(self):
...         return "woof"
...
>>> class DogNotFromObject:
...     def speak(self):
...         return "woof"
...
>>> issubclass(DogFromObject, object)
True
>>> issubclass(DogNotFromObject, object)
False
```

If you don't already base your Python 2 classes on `object`, start today. Old-style classes are long obsolete, and removed in Python 3; they partially or completely break many important tools in Python's object system, like properties and `super()`. The rest of this chapter assumes you're inheriting from `object`.

3.2 Properties

In object-oriented programming, a *property* is a special sort of class member. It's almost a cross between a method and an attribute. The idea is that you can, when designing the class, create "attributes" whose reading, writing, and so on can be managed by special methods. In Python, you do this with a decorator named `property`. Here's an example:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return self.first_name + " " + self.last_name
```

By instantiating this, I can access `full_name` as a kind of virtual attribute:

```
>>> joe = Person("Joe", "Smith")
>>> joe.full_name
'Joe Smith'
```

Notice carefully the members here: there are two attributes called `first_name` and `last_name`, set in the constructor. There is also a method called `full_name`. But after creating the object, we reference `joe.full_name` as an attribute; we don't call `joe.full_name()` as a method.

This is all due to the `@property` decorator. When applied to a method, this decorator makes it inaccessible as a method. You must access it as an attribute. In fact, if you try to call it as a method, you get an error:

```
>>> joe.full_name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

As defined above, `full_name` is read-only. We can't modify it:

```
>>> joe.full_name = "Joseph Smith"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In other words, Python properties are read-only by default. Another way of saying this is that `@property` automatically defines a *getter*, but not a *setter*. If you do want to `full_name` to be writable, here is how you define the setter:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return self.first_name + " " + self.last_name

    @full_name.setter
    def full_name(self, value):
        self.first_name, self.last_name = value.split(" ", 1)
```

This lets us assign to `joe.full_name`:

```
>>> joe = Person("Joe", "Smith")
>>> joe.first_name
'Joe'
>>> joe.last_name
'Smith'
>>> joe.full_name = "Joseph Smith"
>>> joe.first_name
'Joseph'
>>> joe.last_name
'Smith'
```

The first time I saw this, I had all sorts of questions. "Wait, why is `full_name` defined twice? And why is the second decorator named `@full_name`, and what's this `setter` attribute? How on earth does this even byte compile?"

The code is actually correct, and designed to work this way. The `@property def full_name` must come first. That creates the property to begin with, and *also* creates the getter. By "create the property", I mean that an object named `full_name` exists *in the namespace of the class*, and it has a method named `full_name.setter`. This `full_name.setter` is a decorator that is applied to the next `def full_name`, christening it as the setter for the `full_name` property.

It's okay to not fully understand how this all works. A full explanation relies on understanding both implementing decorators, and Python's descriptor protocol, both of which are beyond the scope of what we want to focus on here. Fortunately, you don't have to understand *how* it works in order to use it.

Besides getting and setting, properties have a couple of other options which are less commonly used. `full_name.deleter` can be used as a decorator to handle the `del` operation for the object attribute. This seems to be rarely needed in practice, but it's available when you do.

What you see here with the `Person` class is one way properties are useful: magic attributes whose values are derived from other values. This denormalizes the object's data, and lets you access the property value as an attribute instead of as a method. (The benefit of this is sometimes more cognitive than anything else.)

Properties enable a useful collection of design patterns. One - as mentioned - is in creating read-only member variables. In `Person`, the `full_name` "member variable" is a dynamic attribute; it doesn't exist on its own, but instead calculates its value at run-time.

It's also common to have the property backed by a single, non-public member variable. That pattern looks like this:

```
class Coupon:
    def __init__(self, amount):
        self._amount = amount
    @property
    def amount(self):
        return self._amount
```

This allows the class itself to modify the value internally, but prevent outside code from doing so:

```
>>> coupon = Coupon(1.25)
>>> coupon.amount
1.25
>>> coupon.amount = 1.50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In Python, prefixing a member variable by a single underscore signals the variable is non-public, i.e. it should only be accessed internally, inside methods of that class, or its subclasses.[10] What this pattern says is "you can access this variable, but not change it".

Between "regular member variable" and "ready-only" is another pattern: allow changing the attribute, but validate it first. Let me explain. Suppose my event-management application has a `Ticket` class, representing tickets sold to concert-goers:

```
class Ticket:
    def __init__(self, price):
        self.price = price
    # And some other methods...
```

One day, we find a bug in our web UI, which lets some shifty customers adjust the price to a negative value… so we ended up actually *paying* them to go to the concert. Not good!

The first priority is, of course, to fix the bug in the UI. But do we modify our code to prevent this from ever happening again? Before reading further, look at the `Ticket` class and ponder - how could you use properties to make this kind of bug impossible in the future?

[10] This isn't enforced by Python itself. If your teammates don't already honor this widely-followed convention, you'll have to educate them.

The answer: verify the new price is non-zero in the setter:

```
# Version 1...
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

This lets the price be adjusted... but only to sensible values:

```
>>> t = Ticket(42)
>>> t.price = 24 # This is allowed.
>>> print(t.price)
24
>>> t.price = -1 # This is NOT.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in price
ValueError: Nice try
```

However, there's a defect in this new `Ticket` class. Can you spot what it is? (And how to fix it?)

The problem is that while we can't *change* the price to a negative value, this first version lets us *create* a ticket with a negative price to begin with. That's because we write `self._price = price` in the constructor. The solution is to use the *setter* in the constructor instead:

```
# Final version, with modified constructor.
# (Constructor is different; code for getter & setter is the same ↩
    .)
class Ticket:
    def __init__(self, price):
        # instead of "self._price = price"
        self.price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Yes, you can reference `self.price` in methods of the class. When we write `self.price = price`, Python translates this to calling the `price` setter - i.e., the second `price()` method. This final version of `Ticket` centralizes all reads AND writes of `self._price` in the property. It's a useful encapsulation principle in general. The idea is you centralize any special behavior for that member variable in the getter and setter, even for the class's internal code. In practice, sometimes you will find you need to violate this rule; you simply reference `self._price` and move on. But if you avoid that as long as you can, you will probably benefit from higher quality code.

3.2.1 Properties and Refactoring

Properties are important in most languages today. Here's a situation that often plays out. Imagine writing a simple money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # And some other methods...
```

Suppose you put this class in a library, which many developers are using. People on your current team, perhaps developers on different teams. Or maybe you release it as open-source, so developers around the world use and rely on this class.

Now, one day you realize many of Money's methods - which do calculations on the money amount - can be simpler and more straightforward if they operate on the total number of cents, rather than dollars and cents separately. So you refactor the internal state:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
```

This minor change creates a MAJOR maintainability problem. Can you spot it?

Here's the trouble: your original Money has attributes named dollars and cents. And since many developers are using these, changing to total_cents breaks all their code!

```
money = Money(27, 12)
message = "I have {:d} dollars and {:d} cents."
# This line breaks, because there's no longer
# dollars or cents attributes.
print(message.format(money.dollars, money.cents))
```

If no one but you uses this class, there's no real problem - you can just refactor your own code. But if that's not the case, coordinating this change with everyone's different code bases is a nightmare. It becomes a barrier to improving your own code.

So, what do you do? Can you think of a way to handle this situation?

You get out of this mess is with properties. You want two things to happen:

1. Use total_cents internally, and
2. All code using dollars and cents continues to work, without modification.

You do this by replacing dollars and cents with total_cents internally, but also creating getters and setters for these attributes. Take a look:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
    # Getter and setter for dollars...
    @property
    def dollars(self):
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents = 100 * new_dollars + self.cents
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents = 100 * self.dollars + new_cents
```

Now, I can get and set `dollars` and `cents` all day:

```
>>> money = Money(27, 12)
>>> money.total_cents
2712
>>> money.cents
12
>>> money.dollars = 35
>>> money.total_cents
3512
```

Python's way of doing properties brings many benefits. In languages like Java, the following story often plays out:

1. A newbie developer starts writing Java classes. They want to expose some state, so create public member variables.
2. They use this class everywhere in their code base. Other developers do too.
3. One day, they want to change the name or type of that member variable, or even do away with it entirely (like what we did with `Money`).
4. But that would break everyone's code. So they can't.

Because of this, Java developers quickly learn to make all their variables private by default - proactively creating getters and setters for *every* publicly exposed chunk of data. They realize this boilerplate is far less painful than the alternative, because if everyone must use the public getters and setters to begin with, you always have the freedom to make internal changes later.

This works well enough. But it *is* distracting, and just enough trouble that there's always the temptation to make that member variable public, and be done with it.

In Python, we have the best of both worlds. We make member variables public by default, refactoring them as properties if and when we ever need to. No one using our code even has to know.

3.3 The Factory Patterns

There are several design patterns with the word "factory" in their names. Their unifying idea is providing a handy, simplified way to create useful, potentially complex objects. There two most important forms are:

- Where the object's type is fixed, but we want to have several different ways to create it. This is called the *Simple Factory Pattern*.
- Where the factory dynamically chooses one of several different types. This is called the *Factory Method Pattern*.

Let's look at how you do these in Python.

3.3.1 Alternative Constructors: The Simple Factory

Imagine a simple `Money` class, suitable for currencies which have dollars and cents:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

We looked at this in the previous section, refactoring its attributes - but let's roll back, and focus instead on the constructor's interface. This constructor is convenient when we have the dollars and cents as separate integer variables. But there are many other ways to specify an amount of money. Perhaps you're modeling a giant jar of pennies:

```
# Emptying the penny jar...
total_pennies = 3274
# // is integer division
dollars = total_pennies // 100
cents = total_pennies % 100
total_cash = Money(dollars, cents)
```

Suppose your code splits pennies into dollars and cents over and over, and you're tired of repeating this calculation. You could change the constructor, but that means refactoring all Money-creating code, and perhaps a lot of code fits the current constructor better anyway. Some languages let you define several constructors, but Python makes you pick one.

In this case, you can usefully create a *factory function* taking the arguments you want, creating and returning the object.

```
# Factory function taking a single argument, returning
# an appropriate Money instance.
def money_from_pennies(total_cents):
    dollars = total_cents // 100
    cents = total_cents % 100
    return Money(dollars, cents)
```

Imagine that, in the same code base, you also routinely need to parse a string like "$140.75". Here's another factory function for that:

```
# Another factory function, creating Money from a string amount.
import re
def money_from_string(amount):
    match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', ↩
       amount)
    assert match is not None, 'Invalid amount: {}'.format(amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

These are effectively alternate constructors: callables we can use with different arguments, which are parsed and used to create the final object. But this approach has problems. First, it's awkward to have them as separate functions, defined outside of the class. But much more importantly: what happens if you subclass Money? Suddenly money_from_string and money_from_pennies are worthless. The base Money class is hard-coded.

Python solves these problems in unique way, absent from other languages: the `classmethod` decorator. Use it like this:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
```

The function `money_from_pennies` is now a method of the `Money` class, called `from_pennies`. But it has a new argument: `cls`. When applied to a method definition, `classmethod` modifies how that method is invoked and interpreted. The first argument is not `self`, which would be an *instance* of the class. The first argument is now *the class itself*. In the method body, `self` isn't mentioned at all; instead, `cls` is a variable holding the current class object - `Money` in this case. So the last line is creating a new instance of `Money`:

```
>>> piggie_bank_cash = Money.from_pennies(3217)
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
32
>>> piggie_bank_cash.cents
17
```

Notice `from_pennies` is invoked off the class itself, not an instance of the class. This already is nicer code organization. But the real benefit is with inheritance:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

This is the *real* benefit of class methods. You define it once on the base class, and all subclasses can leverage it, substituting their own type for `cls`. **This makes class methods perfect for the**

simple factory in Python. Here's the complete Money class, with

Notice `self` is not mentioned anywhere in the body. The final line returns an instance of `cls`, using its regular constructor. Here, `cls` is `Money`, so that last line is exactly equivalent to the freestanding function version above.

For the record, here's how we translate `money_from_string`:

```python
    @classmethod
    def from_string(cls, amount):
        match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
        assert match is not None, 'Invalid amount: {}'.format(amount)
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

Class methods are a superior way to implement factories like this in Python. If we subclass `Money`, that subclass will have `from_pennies` and `from_string` methods that create objects of that subclass, without any extra work on our part. And if we change the name of the `Money` class, we only have to change it in one place, not three.

This form of the factory pattern is called "simple factory", a name I don't love. I prefer to call it "alternate constructor". Especially in the context of Python, it describes well what `@classmethod` is most useful for. And it suggests a general principle for designing your classes. Look at this complete code of the `Money` class, and I'll explain:

```python
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
    @classmethod
    def from_string(cls, amount):
        import re
        match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$ ↩
            ', amount)
        if match is None:
            raise ValueError('Invalid amount: {}'.format(amount))
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
    # And then some other methods...
```

You can think of this class as having several constructors. As a general rule, you'll want to make `__init__` the most generic one, and implement the others as class methods. Sometimes, that means one of the class methods will be used more often than `__init__`. This goes against the intuitions of many Python developers; if that describes your team, you'll want to educate them in the class docs.

3.3.2 Dynamic Type: The Factory Method Pattern

This next factory pattern, called "Factory Method", is quite different. The idea is that the factory will create an object, but will choose its type from one of several possibilities, dynamically deciding at run-time based on some criteria. It's typically used when you have one base class, and are creating an object that can be one of several different derived classes.

Let's see an example. Imagine you are implementing an image processing library, creating classes to read the image from storage. So you create a base `ImageReader` class, and several derived types:

```
import abc
class ImageReader(metaclass=abc.ABCMeta):
    def __init__(self, path):
        self.path = path
    @abc.abstractmethod
    def read(self):
        pass # Subclass must implement.

    def __repr__(self):
        return '{}({})'.format(self.__class__.__name__, self.path ↩
           )

class GIFReader(ImageReader):
    def read(self):
        "Read a GIF"

class JPEGReader(ImageReader):
    def read(self):
        "Read a JPEG"

class PNGReader(ImageReader):
    def read(self):
        "Read a PNG"
```

The `ImageReader` class is marked abstract, requiring subclasses to implement the `read` method. So far, so good.

Now, when reading an image file, if its extension is ".gif", I want to use `GIFReader`. And if it is a JPEG image, I want to use `JPEGReader`. And so on. The logic is

- Analyze the file path name to get the extension,
- choose the correct reader class based on that,
- and finally create the appropriate reader object.

This is a prime candidate for automation. Let's define a little helper function:

```
def extension_of(path):
    position_of_last_dot = path.rfind('.')
    return path[position_of_last_dot+1:]
```

With these pieces, we can now define the factory:

```
# First version of get_image_reader().
def get_image_reader(path):
    image_type = extension_of(path)
    reader_class = None
    if image_type == 'gif':
        reader_class = GIFReader
    elif image_type == 'jpg':
        reader_class = JPEGReader
    elif image_type == 'png':
        reader_class = PNGReader
    assert reader_class is not None, 'Unknown extension: {}'. ↩
       format(image_type)
    return reader_class(path)
```

Classes in Python can be put in variables, just like any other object. We take full advantage here, by storing the appropriate `ImageReader` subclass in `reader_class`. Once we decide on the proper value, the last line creates and returns the reader object.

This correctly-working code is more concise, readable and maintainable than what some languages force you to go through. But in Python, we can do better. We can use the built-in dictionary type to make it even more readable and easy to maintain over time:

```
READERS = {
    'gif' : GIFReader,
    'jpg' : JPEGReader,
    'png' : PNGReader,
    }
def get_image_reader(path):
    reader_class = READERS[extension_of(path)]
    return reader_class(path)
```

Here we have a global variable mapping filename extensions to `ImageReader` subclasses. This lets us readably implement `get_image_reader` in two lines. Finding the correct class is a simple dictionary lookup, and then we instantiate and return the object. And if we support new image formats in the future, we simply add a line in the `READERS` definition. (And, of course, define its reader class.)

What if we encounter an extension not in the mapping, like `tiff`? As written above, the code will raise a `KeyError`. That may be what we want. Or closely related, perhaps we want to catch that, and re-raise a different exception.

Alternatively, we may want to fall back on some default. Let's create a new reader class, meant as an all-purpose fallback:

```
class RawByteReader(ImageReader):
    def read(self):
        "Read raw bytes"
```

Then you can write the factory like:

```
def get_image_reader(path):
    try:
        reader_class = READERS[extension_of(path)]
    except KeyError:
        reader_class = RawByteReader
    return reader_class(path)
```

or more briefly

```
def get_image_reader(path):
    return READERS.get(extension_of(path), RawByteReader)
```

This design pattern is commonly called the "factory method" pattern, which wins my award for Worst Design Pattern Name In History. That name (which appears to originate from a Java implementation detail) fails to tell you anything about what it's actually *for*. I myself call it the "dynamic type" pattern, which I feel is much more descriptive and useful.

3.4 The Observer Pattern

The Observer pattern defines a certain kind of "one to many" relationship. Specifically, there's one root object - let's call it the *publisher* - whose state can change in a way that's interesting to other objects. These other objects - let's call them *subscribers* - tell the publisher that they want to know when this happens.

The way they tell the publisher is to *register* (by calling a method on the publisher, which may actually be named `register`). Whenever this interesting state in the publisher changes, all registered subscribers are notified.

That's all pretty abstract. Let's see some concrete examples.

3.4.1 The Simple Observer

In the simplest form, each subscriber must implement a method called `update` (or something else that both sides agree on). Here's an example:

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)
```

The `update` method takes a string. (It can take something else - again, so long as both publisher and subscriber agree on the interface. But we'll use a string.)

Example driver:

```python
from observer1 import Publisher, Subscriber

pub = Publisher()

bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register(bob)
pub.register(alice)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

3.4.2 A Pythonic Refinement

The simple version above has a pretty standard interface. Python gives more flexibility, because you can pass functions around just like any other object. This means a subscriber can register to be notified by calling a method other than `update` - or even a completely separate function.

Regardless of whether this is a method or a function, let's just call it a "callback". This callback should have a compatible signature, of course.

```python
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback == None:
            callback = getattr(who, 'update')
        self.subscribers[who] = callback
    def unregister(self, who):
        del self.subscribers[who]
    def dispatch(self, message):
        for subscriber, callback in self.subscribers.items():
            callback(message)
```

```python
from observer2 import *

pub = Publisher()
bob = SubscriberOne('Bob')
alice = SubscriberTwo('Alice')
john = SubscriberOne('John')

pub.register(bob, bob.update)
pub.register(alice, alice.receive)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

3.4.3 Several Channels

So far, we've assumed the publisher only has one kind of thing to say… one kind of state that can change, which is of interest to subscribers. But what if there are several?

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self, channels):
        # maps channel names to subscribers
        # str -> dict
        self.channels = { channel : dict()
                          for channel in channels }
    def subscribers(self, channel):
        return self.channels[channel]
    def register(self, channel, who, callback=None):
        if callback == None:
            callback = getattr(who, 'update')
        self.subscribers(channel)[who] = callback
    def unregister(self, channel, who):
        del self.subscribers(channel)[who]
    def dispatch(self, channel, message):
        for subscriber, callback in self.subscribers(channel). ↩
           items():
            callback(message)
```

```
from observer3 import *

pub = Publisher(['lunch', 'dinner'])
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)

pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

3.5 Magic Methods

Suppose we want to create a class to work with angles, in degrees. We want this class to help us with some standard bookkeeping:

- An angle will be at least zero, but less than 360.
- If we create an angle outside this range, it automatically wraps around to an equivalent, in-range value.
- In fact, we want the conversion to happen in a range of situations:
  - If we add 270º and 270º, it evaluates to 180º instead of 540º.
  - If we subtract 180º from 90º, it evaluates to 270º instead of -90º.
  - If we multiply an angle by a real number, it wraps the final value into the correct range.
- And while we're at it, we want to enable all the other behaviors we normally want with numbers: comparisons like "less than" and "greater or equal than" or "==" (i.e., equals); division (which doesn't normally require casting into a valid range, if you think about it); and so on.

Let's see how we might approach this, by creating a basic Angle class:

```
class Angle:
    def __init__(self, value):
        self.value = value % 360
```

The modular division in the constructor is kind of neat: if you reason through it with a few positive and negative values, you'll find the math works out correctly whether the angle is overshooting or undershooting. This meets one of our key criteria already: the angle is normalized to be from 0 up to 360. But how do we handle addition? We of course get an error if we try it directly:

```
>>> Angle(30) + Angle(45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Angle' and 'Angle'
>>>
```

We can easily define a method called `add` or something, which will let us write code like `angle3 = angle1.add(angle2)`. But that means we can't reuse the familiar syntax everyone learned in school for math. I don't know about you, but when I'm hard at work developing, I already have more than enough to learn and remember. So when possible, I prefer to rely on something so ingrained and automatic, that I'm free to focus my mental energy on more important things.

Well, great news: Python lets us do that, through a collection of object hooks called *magic methods*. It lets you define classes so that their instances can be used with all of Python's primitive operators:

- You can do all arithmetic using the normal operators: + - * / // and more
- They can be compared for equality (==) and inequality (!=)
- ... as well as richer comparisons (< > >= <=)
- Higher-level concepts like exponentiation, absolute value, etc.
- Bit-shifting operations

Few classes will need all of these, but sometimes it's invaluable to have them available. Let's see how they can improve our `Angle` type.

3.5.1 Simple Math Magic

The pattern for each method is the same. For a given operation - say, addition - there is a special method name that starts and begins with double-underscores. For addition, it's `__add__` - the others also have sensible names. All you have to do is define that method, and instances of your class can be used with that operator.

For operations like addition that take two values and return a third, the signature looks like this:

```
    def __add__(self, other):
        return 42 # Or whatever the correct total is.
```

The first argument needs to be called "self", because this is Python. The other one does not have to be called "other", but often it is, because it has a clear meaning and is easy to remember. For our `Angle` class, we could implement it like this:

```
    def __add__(self, other):
        return Angle(self.value + other.value)
```

This lets us use the normal addition operator for arithmetic:

```
>>> total = Angle(30) + Angle(45)
>>> total.value
75
```

There are similar operators for subtraction (`__sub__`), multiplication (`__mul__`), and more.

3.5.2 Printing and Logging

There's something missing, though: what if we try to add two angles directly, without setting to a variable?

```
>>> Angle(30) + Angle(45)
<__main__.Angle object at 0x106df9198>
```

The `__add__` method is returning a correct object. But when we print it, the representation isn't so useful. It tells us the type, and the hex object ID. But what we might really want to know is the value of the angle.

There are two magic methods that can help. The first is `__str__`, which is used when printing a result:

```
    def __str__(self):
        return '{} degrees'.format(self.value)
```

The `print` function uses this, as well as `str`, and the string formatting operations:

```
>>> print(Angle(30) + Angle(45))
75 degrees
>>> print('{}'.format(Angle(30) + Angle(45)))
75 degrees
>>> str(Angle(135))
'135 degrees'
```

Sometimes, you want a string representation that is more precise, which might be at odds with the goals of a human-friendly representation. A good example is when you have several subclasses (e.g., imagine `PitchAngle` and `YawAngle` in some kind of aircraft-related library), and want to easily log the exact type and arguments needed to recreate the object. Python provides a second magic method for this purpose, called `__repr__`:

```
    # An okay implementation.
    def __repr__(self):
        return 'Angle({})'.format(self.value)
```

You access this by calling either the `repr` built-in function (think of it as working like `str`, but invokes `__repr__` instead of `__str__`), or by passing the `!r` conversion to the formatting string:

```
>>> print('{!r}'.format(Angle(30) + Angle(45)))
Angle(75)
```

The rule of thumb is that the output of `__repr__` is something that can be passed to `eval()` to recreate the object exactly. While not enforced by the language, this convention is officially recommended, and very widely followed among experienced Python programmers. It's not always practical for every class. But often it is, and it can be very useful for logging and other purposes.

3.5.3 All Things Being Equal

Another thing we want to be able to do is compare two `Angle` objects. The most basic is equality and inequality. The former is provided by `__eq__`, which should return True or False:

```
    def __eq__(self, other):
        return self.value == other.value
```

If defined, this method is used by the == operator to determine equality:

```
>>> Angle(3) == Angle(3)
True
>>> Angle(7) == Angle(1)
False
```

By default, the == operator for objects is based off the object ID, which is safe but often not very useful:

```
>>> class BadAngle:
...     def __init__(self, value):
...         self.value = value
...
>>> BadAngle(3) == BadAngle(3)
False
```

The `!=` operator has its own magic method, `__ne__`. It works the same way:

```
    def __ne__(self, other):
        return self.value != other.value
```

What happens if you don't implement `__ne__`? In Python 3, if you define `__eq__` but not `__ne__`, then the `!=` operator will use `__eq__`, negating the output. Especially for simple classes like `Angle`, this default behavior is logically valid. So in this case, we don't need to define a `__ne__` method at all. For more complex types, the concepts of equality and inequality may have more subtle nuances, and you will need to implement both.

3.5.3.1 Python 2 != Python 3

The story is different in Python 2. In that universe, if `__eq__` is defined but `__ne__` is not, then `!=` does *not* use `__eq__`. Instead, it relies on the default comparison based on object ID:

```
# Python 2.
>>> class BadAngle(object):
...     def __init__(self, value):
...         self.value = value
...     def __eq__(self, other):
...         return self.value == other.value
...
>>> BadAngle(3) != BadAngle(3)
True
```

You will probably never actually want this behavior (which is why it was changed in Python 3). So for Python 2, if you do define `__eq__`, be sure to define `__ne__` also:

```
# A good default __ne__ for Python 2.
# This is basically what Python 3 does automatically.
    def __ne__(self, other):
        return not self.__eq__(other)
```

3.5.4 Comparisons

Now that we've covered strict equality and inequality, what's left are the fuzzier comparison operations; less than, greater than, and so on. Python's documentation calls these "rich comparison" methods, so you can feel wealthy when using them:

- `__lt__` for "less than" (<)
- `__le__` for "less than or equal" (<=)
- `__gt__` for "greater than" (>)
- `__ge__` for "greater than or equal" (>=)

For example:

```
    def __gt__(self, other):
        return self.value > other.value
```

Now the greater-than operator works correctly:

```
>>> Angle(100) > Angle(50)
True
```

Similar with `__ge__`, `__lt__`, etc. If you don't define these, you get an error, at least in Python 3:

```
>>> BadAngle(8) > BadAngle(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: BadAngle() > BadAngle()
```

`__gt__` and `__lt__` are reflections of each other. What that means is that, in many cases, you only have to define one of them. Suppose you implement `__gt__` but not `__lt__`, then do this:

```
>>> a1 = Angle(3)
>>> a2 = Angle(7)
>>> a1 < a2
True
```

This works thanks to some just-in-time introspection the Python runtime does. The `a1 < a2` is, semantically, equivalent to `a1.__lt__(a2)`. If `Angle.__lt__` is indeed defined, that semantic equivalent is executed, and the expression evaluates to its return value.

For normal scalar numbers, `n < m` is true if and only if `m > n`. For this reason, if `__lt__` does not exist, but `__gt__` does, then Python will rewrite the angle comparison: `a1.lt(a2)` becomes `a2.gt(a1)`. This is then evaluated, and the expression `a1 < a2` is set to its return value.

Note there are situations where this is actually *not* what you want. Imagine a `Point` type, for example, with two coordinates, x and y. You want `point1 < point2` to be `True` if and only if `point1.x < point2.x`, AND `point1.y < point2.y`. Similarly for `point1 > point2`. There are many points for which both `point1 < point2` and `point1 > point2` should both evaluate to `False`.

For types like this, you will want to implement both `__gt__` and `__lt__`. (Ditto for `__ge__` and `__le__`.) You might also need to raise `NotImplemented` in the method. This built-in exception signals to the Python runtime that the operation is not supported, at least for these arguments.

3.5.4.1 Shortcut: functools.total_ordering

The `functools` module in the standard library defines a class decorator named `total_ordering`. In practice, for any class which needs to implement all the rich comparison operations, using this labor-saving decorator should be your first choice.

In essence: in your class, you define both `__eq__` and **one** of the comparison magic methods: `__lt__`, `__le__`, `__gt__`, or `__ge__`. (You can define more than one, but it's not necessary.) Then you apply the decorator to the *class*:

```
import functools
@functools.total_ordering
class Angle:
    # ...
    def __eq__(self, other):
        return self.value == other.value
    def __gt__(self, other):
        return self.value > other.value
```

When you do this, all missing rich comparison operators are supplied, defined in terms of `__eq__` and the one operator you defined. This can save you a fair amount of typing, and it's worthwhile to use it if it makes sense.

There are a few situations where you won't want to use `total_ordering`. One is if the comparison logic for the type is not well-behaved enough that each operator can be inferred from the other, via straightforward boolean logic. The `Point` class is an example of this, as might some types if what you are implementing boils down to some kind of abstract algebra engine.

The other reasons not to use it are (1) performance, and (2) the more complex stack traces it generates are more trouble than they are worth. Generally, though, I recommend you assume these are *not* a problem until proven otherwise. It's entirely possible you will never encounter one of the involved stack traces. And the relatively inefficient implementations that `total_ordering` provides are unlikely to be a problem unless they are used deep inside some nested loop.

3.5.5 Python 2

As mentioned, in Python 3, if you don't define `__lt__`, and then try to compare two objects with the < operator, you get a `TypeError`. And likewise for `__gt__` and the others. That's a *very* good thing. In Python 2, you instead get a default ordering based off the object ID. This can lead to truly infuriating bugs:

```
# Python 2.
>>> class BadAngle(object):
...     def __init__(self, value):
...         self.value = value
...
>>>
>>> BadAngle(6) < BadAngle(5)
True
>>> BadAngle(6) < BadAngle(5)
False
```

What the heck just happened? When parsing and running the first `BadAngle(6) < BadAngle(5)` line, the Python runtime created two `BadAngle` instances. It turns out the left-hand object was created with an ID whose value happens to be less than that of the right-hand object. So the expression evaluates as `True`. In the second line, the opposite happened: the right-hand object won the race, so to speak, so the expression evaluates as `False`.

Horrifying race conditions like this are not your friend. Until you can upgrade to Python 3, be vigilant.

3.5.6 More Magic

The full list of numeric magic methods is listed and documented at https://docs.python.org/3/-reference/datamodel.html#emulating-numeric-types. Most of the magic methods we covered relate to numeric operations, but there are others as well. You can read more about them in the surrounding sections. They include:

- Boolean operations, like `and`, `or` and `not`
- Higher-level math operations like `abs`, exponents, and negation
- Bit-shifting operations

Index